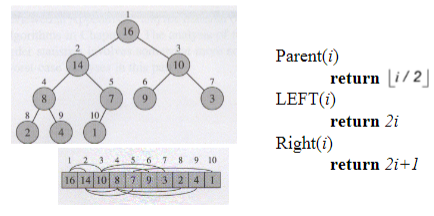
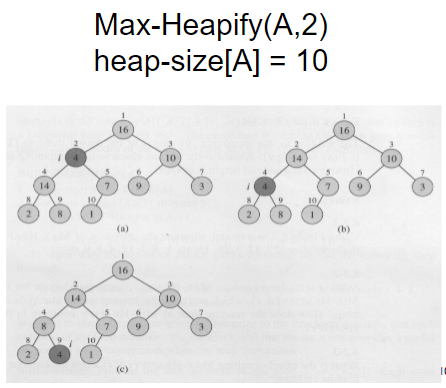
**Binary Heaps:**

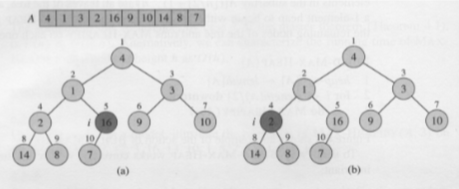
* An array object that can be viewed as a complete tree.

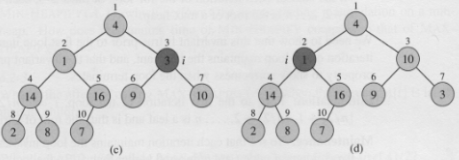


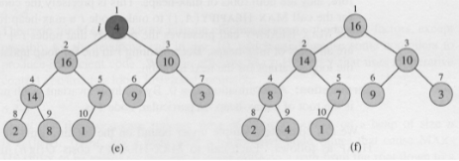
* Heap Property:
  + Max-Heap: A[parent(i)] >= A[i]
  + Min-Heap: A[parent(i)] <= A[i]
  + Height of a Node: The number of edges on the longest downward path from the node to a leaf.
  + Height of a Tree: The height of the root.
  + Height of a Heap: Floor(lgn) = O(lgn)
* Max-Heapify Algorithm: It is an important subroutine for manipulating heaps. Its inputs are an array A and an index *i* in the array. When Heapify is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but that A[i] may be smaller than its children, thus violating the max heap property.



* + Max-Heapify Complexity: It has to be O(height); the height of a heap is O(lgn), so Max-Heapify is O(lgn).
  + Base Case: T(base case) = Ө(1) = c
  + Recurrence Relation: T(n) <= T(2n/3) + Ө(1) or T(2n/3) + c
* Build-Max-Heap(A):
  + Heap-size(A) = length(A)
  + For i = floor(heap-size(A)/2) downto 1
  + Max-Heapify(A, i)
  + Ex:







* + Complexity: O(n)
* Heapsort Algorithm:
  + Heapsort(A):

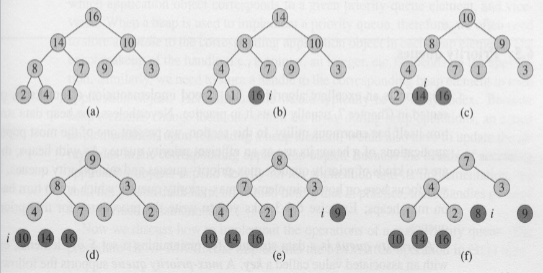
Build-Max-Heap(A)

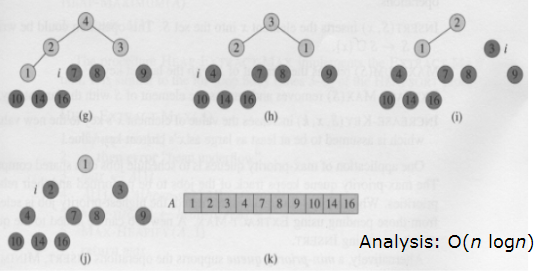
For i = length(A) down to 2

Swap A[1] and A[i]

Heap-size[A] = heap-size[A] – 1

Max-Heapify(A, 1)

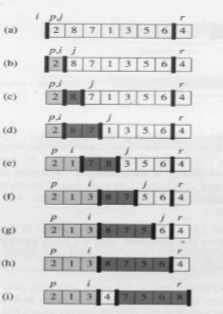




* Priority Queues: The heap data structure is not only used in sorting but also in priority queues. A priority queue is a data structure that maintains a set S of elements, each with an associated value call a key. Priority queues has many applications. A max (or min) priority queue should at least support the following operations:
  + Insert(S, x) O(logn)
  + Maximum/Minimum(S) O(1)
  + Extract-Max/Min(S) O(logn)
  + Increase-Key(S, x, k) O(logn)
  + Decrease-Key(S, x, k) O(logn)

**Quicksort:**

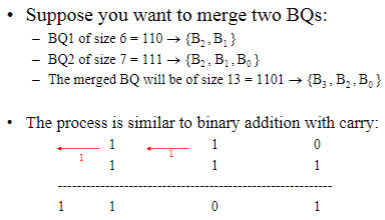
* Objective: Divide, conquer, and combine
* Partition Method:



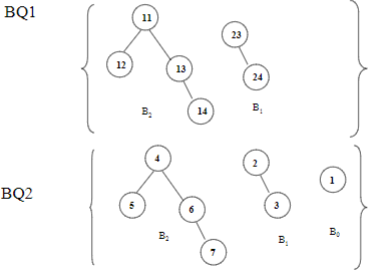
* Loop Invariant: At the beginning of any iteration of the loop of lines 3-6 with a j value between p and r – 1, for any array index k,
  + if p <= k <= i, then A[k] <= x
  + if i + 1 <= k <= j – 1, then A[k] > x
  + if k = r, then A[k] = x
  + Only two cases possible for what happens to A[j] in any one iteration of procedure Partition.
* Quicksort Recurrences:
  + T(n) = T(size of left partition) + T(size of right partition) + Ө(n) = T(0) + T(n – 1) + Ө(n) = T(n – 1) + Ө(n) = T(n – 1) = cn
  + T(1) = Ө(1) = c
  + You can easily show by backward substitution method that these recurrences have the solution T(n) = Ө(n2)
  + Partitioning can also divide the array equally: one partition of size floor(n/2) and the other of size ceiling(n/2) – 1
  + T(n) <= 2T(n/2) + Ө(n)
  + T(1) = Ө(1)
  + You can easily show by applying the master method that if T(n) = 2T(n/2) + Ө(n) then T(n) = Ө(nlgn). So in this case, Quicksort is O(nlgn).
* Average Case Performance: Good and bad splits tend to balance out in practice, so the average performance of quicksort is also O(nlgn).
  + To get this balance, in practice we don’t pick A[r] as the pivot; instead a median-of-three approach is used to pick the pivot.
  + Another way to make sure of random distribution of good and bad splits is to choose randomly so that any of the r-p+1 elements in the array has an equal chance of being picked.
* Comparison Sorts:
  + Sorting by comparing pairs of numbers
  + Algorithms that sort n numbers in O(n2) time – insertion, selection and bubble
  + Algorithms that sort n numbers in O(nlgn) time – Merge, Heap and Quick Sort
  + Any comparison sort must make Ω(nlgn) comparisons, so nlgn is the most efficient they can be!
* Counting Sort: See question three in assignment 3 for a simple example that pretty much sums up how to do it (NOTE: the sorted array will always be perfectly sorted at the end, so just worry about the counting array).
  + Meant to be used when every number being sorted is a whole number
* Radix Sort: See question four in assignment 3 for a simple example that pretty much sums up how to do it.
  + Meant to be used when every number being sorted has the same amount of digits
* Bucket Sort: See question five in assignment 3 for a simple example that pretty much sums up how to do it.
  + Meant to be used when every number being sorted is a decimal

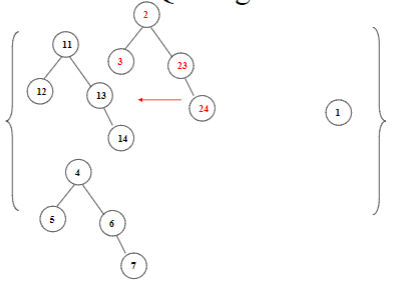
**Binomial Queues/Heaps:**

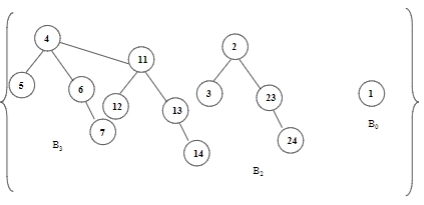
* Consists of a forest of Binomial Trees (BiT); each BiT is a heap
  + Forest: A collection of heap-ordered trees
  + Each tree is a Binomial (not Binary!) Tree
  + At most one Binomial Tree (BiT) of any height in a BQ
  + So a BQ is a collection of trees and satisfies the Heap property and the Structural property
* BiT:
  + Bk: Bit of height k, k=0,1,2
    - B0 = 1-node tree, B1 = 2-node tree, B2 = 4-node tree, …
    - Bi = 2i-node tree
  + You construct a Bk tree by attaching a Bk-1 tree to the root of another Bk-1 tree, making sure that the Heap Property is preserved
  + A Bk tree therefore consists of a root with k child subtrees: B0, B1, …, Bk-1 trees
  + Number of nodes at depth d in a Bk tree is given by the “binomial coefficient” k!/d!(k-d)!; hence the name Binomial Tree
* BQ:
  + A priority queue of any size n can be uniquely represented by a BQ of size n
  + To see how many and which BiTs are in the BQ, look at the binary representation of n
  + This means that there will be floor(logn) + 1 (i.e., O(logn)) Binomial Trees in a BQ of size n
  + Operations:
    - Merge: The most fundamental operation; can be done in O(logn) worst case time.
      * Merging two Bk trees to get a B­k+1 tree is O(1)
      * Merging two BQs is conceptually similar to binary addition of the two numbers representing the sizes of the two BQs to be merged.



* + - * Ex:



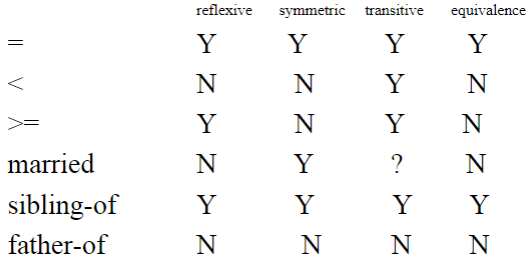




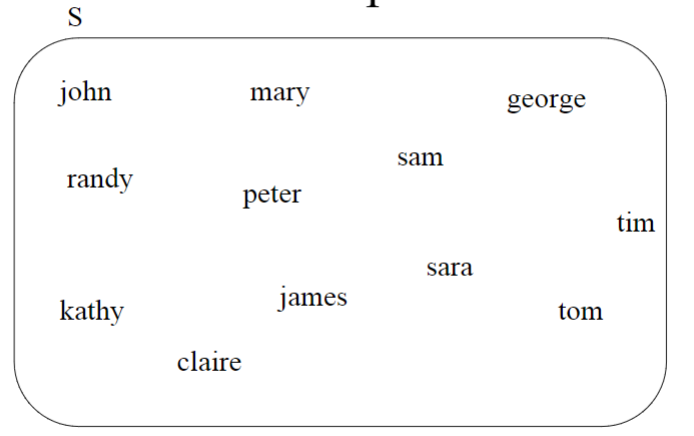
* + - Insert(x, BQ): Merge (BQ’ containing a B0 tree, BQ); also O(logn) worst case
    - Extract-Min(BQ): O(logn) worst case
      * Scan roots of all BiTs in BQ to find the minimum root – O(logn) worst case
      * Delete the root and eventually return the data – O(1)
      * BQ1=BQ containing all the child subtrees of the deleted root
      * BQ2=BQ containing all the other BiTs from BQ
      * Merge BQ1 and BQ2 – O(logn) worst case

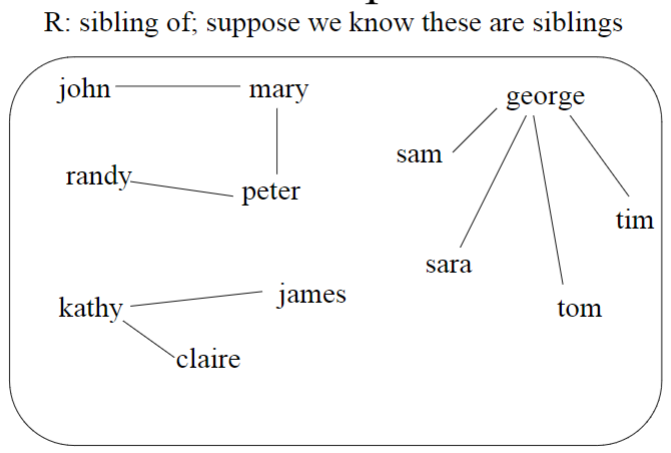
**Disjoint Sets:**

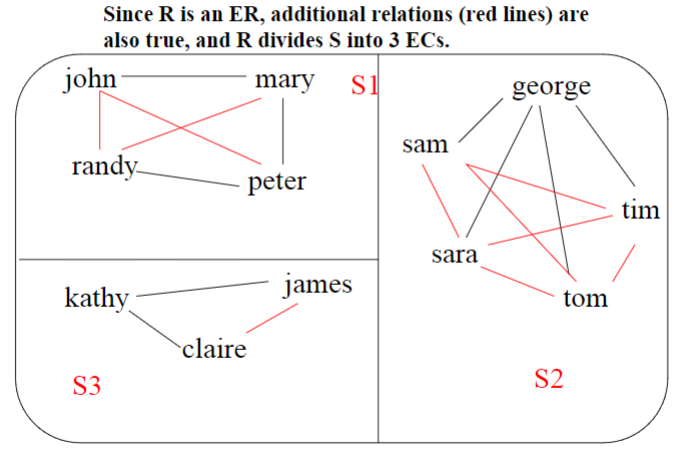
* Relations:
  + R: A relation defined on a set S of elements
  + aRb is true (where a,b є S) => a is related to b by R
  + R is reflexive if aRa is true for all a є S
  + R is symmetric if (aRb is true ⬄ bRa is true) for all a,b є S
  + R is transitive if (aRb is true and bRc is true => aRc is true) for all a,b,c є S
  + R is an Equivalence Relation (ER) if it is reflexive, symmetric and transitive
  + Properties:



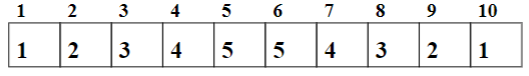
* + Equivalence Problem: If R is an ER defined over a set of objects S, for every pair (a, b) where a,b є S, determine if aRb is true or false
  + Equivalence Class (EC): The EC of an element a є S is the subset of S containing all elements that are related to a by the ER R.
    - In other words, if we define an ER R over a set S, R partitions S into a number of disjoint sets that are ECs.
    - Ex:

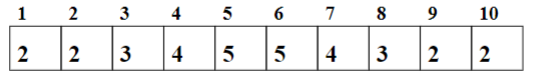




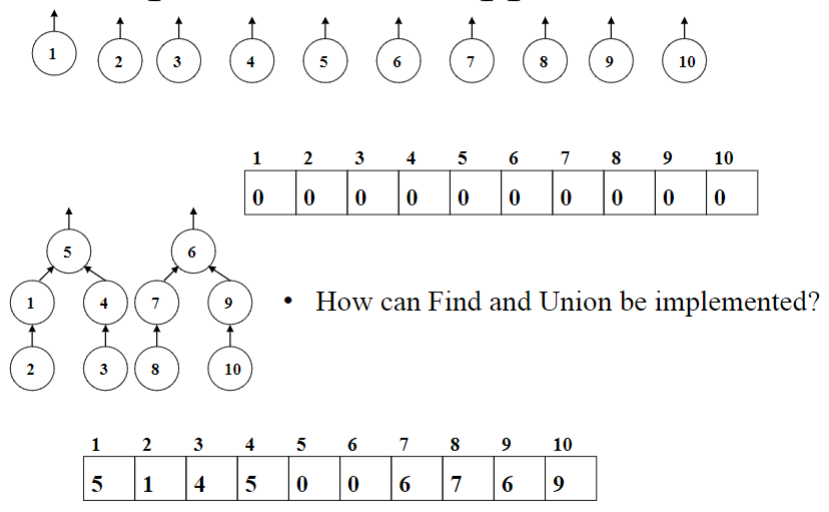


* + - Now, if we maintain information about all ECs in a database of objects with a relation R defined over these objects, we can answer the question aRb? By checking if a and b belong to the same EC.
    - Similarly, we can assert aRb in the database by putting a and b in the same EC; this means that we need to be able to efficiently implement two operations:
      * Determine if aRb is true/false (true if EC(a)=EC(b); false otherwise)
      * Assert aRb in the database (if EC(a)!-EC(b) then merge EC(a) & EC(b))
      * So, we need two operations – Find(element) (returns its EC) and Union(EC1, EC2) (merges the two ECs)
    - Implementation:
      * Data elements in the database are numbered 1…n (ECs are also named this way)
      * Initially we assume that no element is related to any other, i.e., each is in an EC by itself.
      * Two approaches to implementing such a database so that Find and Union operations can be done very efficiently.
    - Implementation Approach 1: Use an array A in which A[i] contains the “name” of the EC to which element i belongs.
      * Find(i) (Where i is an element): Returns A[i]; has Ө(1) time
      * Union(i, j) (Where i and j are names of ECs): Goes through the array and changes all i’s to j’s or vice versa; has Ө(n) time.
        + Ex: First array is before, second array is after Union(1, 2)

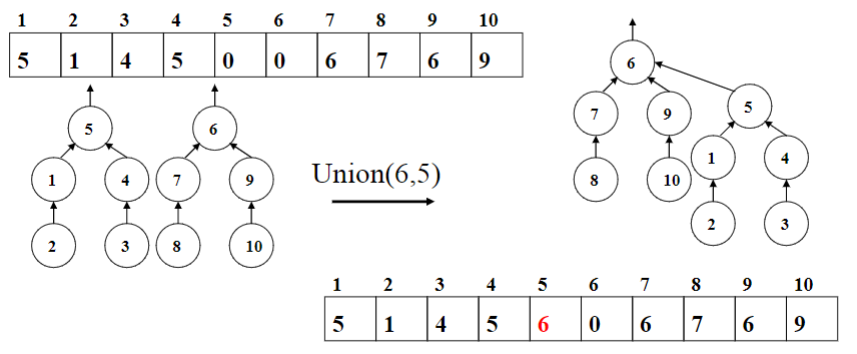




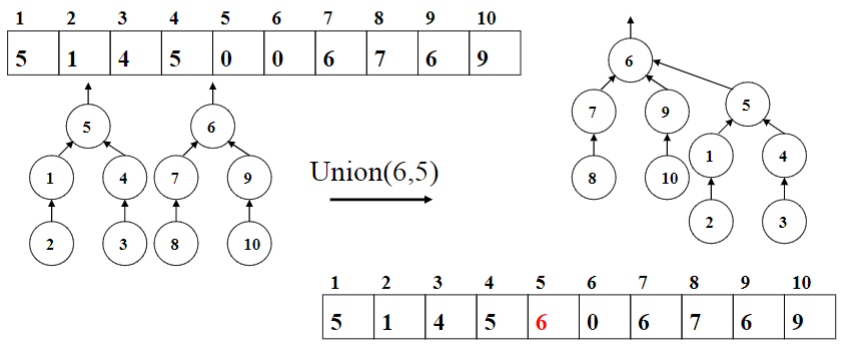
* + - Implementation Approach 2: Keep all elements in the same EC in a tree; the root provides the name of the EC.
      * This generates a forest of trees, which is implemented with an array P.
      * P[i] = parent of i if i is not the root; else P[i] = 0



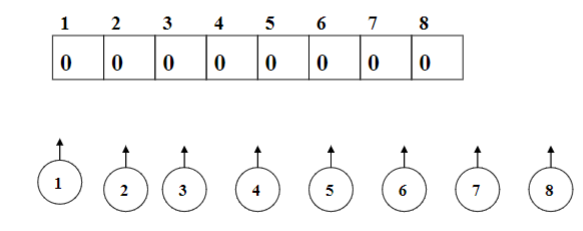
* + - * Find(element): Return the root of the tree containing element; needs exactly as many steps as the depth of element i in its tree; has O(n) time.
      * Union(i, j (Where i & j are roots of trees [names of ECs]): If i != j then set P[i] = j OR P[j] = i (be consistent)

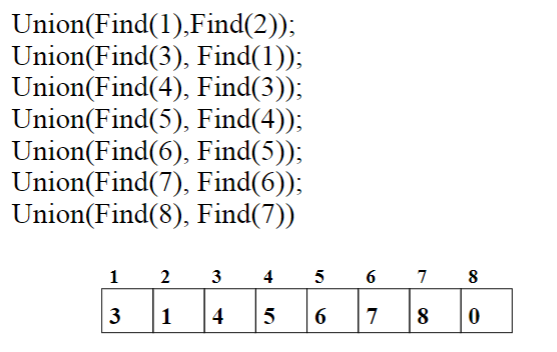
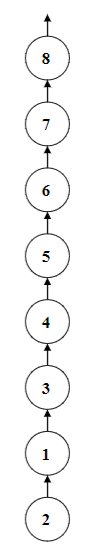


* + - Which Approach is Better:
      * Approach 1: Find is Ө(1) and Union is Ө(n)
      * Approach 2: Find is O(n) and Union is Ө(1); O(n) is better than Ө(n) in this case; so this is the **preferred approach**.
      * In either, a sequence of m operations on a database of size n will require O(mn) time.
* Disjoint Set Operations: Now, let us take a closer look at the fundamental operations Find and Union under the implementation approach 2 where a disjoint set is implemented as a forest.
  + Note that if there are n data elements, the id’s of the elements, names of the disjoint sets, and the indexes of the array implementation range from 1 to n.
  + Union:
    - Arbitrary Union:
      * Union(i, j) (Where i & j Are Roots of Trees [names of ECs]): If i != j then set P[i] = j OR P[j] = i (be consistent); complexity is Ө(1) at best, worst and average case.

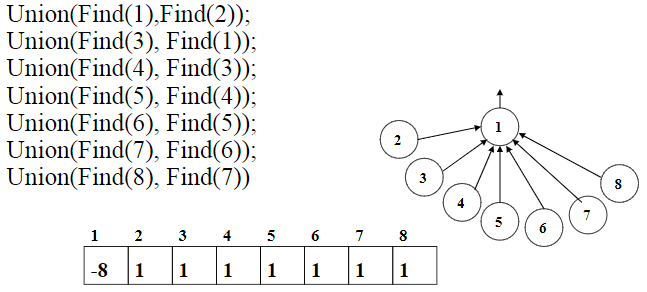


* + - * With arbitrary union, n-deep trees may be generated.
      * Ex:

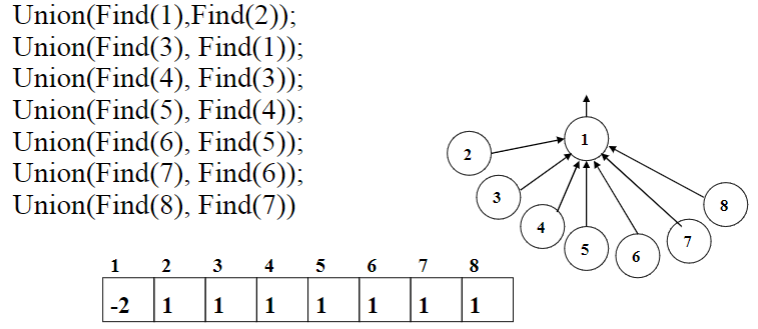


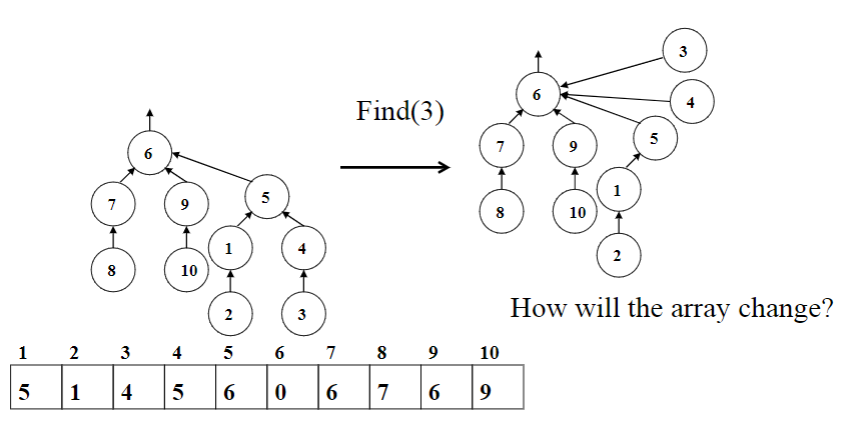
* + “Smart” Union: Use some “smarts” in deciding how to merge two trees; Union-by-Size and Union-by-Height are two different ways
    - Union-by-Size: Make the smaller tree a subtree of the root of the larger one.
      * Implementation: At array cells corresponding to roots, use the negative of tree-size instead of 0.
      * Add the two tree sizes when unioning
      * Previous Example Redo:



* + - * If you start with n nodes in the set originally, any tree resulting from Union-by-Size will have a depth of at most logn.
      * Any tree resulting from Union-by-Size will have a depth <= logn; this implies that Find is O(logn) (an improvement over O(n) with arbitrary Union!)
      * Union is still constant time – Ө(1)
      * A sequence of m operations on a data set of size n has complexity O(mlogn)
    - Union-by-Height: Make the shallower tree a subtree of the root of the deeper one
      * Implementation: At array cells corresponding to roots, use the negative of tree-depth instead of 0
      * Increase the height by 1 when unioning two trees of the same height
      * Previous Example Redo:



* + - * If you start with n nodes in the set originally, any tree resulting from Union-by-Height will have a depth of at most logn.
  + Find:
    - Path Compression: With smart unions we can bound the depth of any tree to logn; but Find operations on leaf nodes will still take logn time.
      * We can improve this situation by modifying the Find operation; during Find(x), change the parent of every node between x and the root to be the root of the tree.
      * An example of **self-adjusting data structure**
      * It gradually decreases the depth of the tree, and items once accessed are generally more likely to be accessed again, and this makes such accesses single-step
      * Works very well with Union-by-Size
      * As PC changes tree heights, for Union-by-Height to work correctly, tree heights need to be recomputed after every Find; this is computationally expensive and cancels out the efficiency achieved by PC, so it does not work very well with Union-by-Height
        + So, the most efficient operations can be achieved by Find-with-PC and Union-by-Size
    - Find With Path Compression Ex:



* + - Find Without Path Compression Ex:

